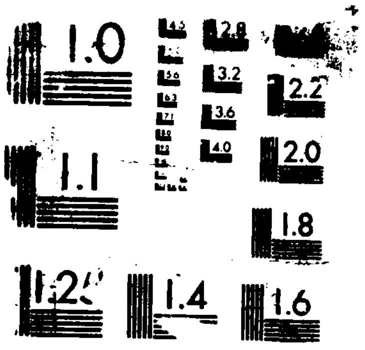
AD-A188 351   EQL'S USER'S GUIDE(U) NORTH CAROLINA UNIV AT CHAPEL   1/1
HILL DEPT OF COMPUTER SCIENCE B JAYARAMAN ET AL
SEP 87 TR87-010-REV N00014-86-K-00680

UNCLASSIFIED   F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART

4

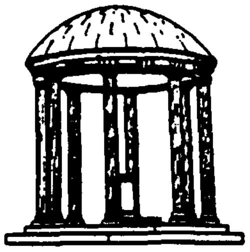
EqL's User's Guide

TR87-010

May, 1987 (Revised September, 1987)

Contract N00014-86-K-0680

Bharat Jayaraman, Gopal Gupta



The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27514

EqL User's Guide[†]

Bharat Jayaraman
Gopal Gupta

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract

(Equational Languages)

EqL is a general-purpose language that combines the capabilities of functional and logic programming languages. A program in EqL consists of a collection of conditional, pattern-directed rules, where the conditions are expressed as a conjunction of equations, and the patterns are terms built up of data-constructors and basic values. The computational paradigm in EqL is equation solving. ~~In~~ this report ~~we~~ describes EqL informally, by first presenting the syntax of constructs and the built-in operations, and then showing how to write and run programs using the EqL interpreter. Several examples are presented, illustrating the various features of the language: nondeterminism, logical variables, deferred evaluation of primitives, higher-order operations, and user-defined constructors. The report also describes I/O operations and other features of the interpreter, including program tracing.

(Keywords: functional programming; logic programming, debugging).

† This research is supported by grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

I. Introduction

EqL, for Equational Language, is a general-purpose programming language that combines the capabilities of two closely-related declarative languages: functional and logic. A program in EqL consists of a collection of conditional, pattern-directed rules, where the conditions are expressed as a conjunction of equations, and the patterns are terms built up of data-constructors, numbers, booleans, atoms, and variables. The theory and rationale behind EqL is discussed elsewhere [J85, JS86, J87]; in this document we describe EqL informally, by first presenting the syntax of constructs and the built-in operations, and then showing how to write and run programs using the EqL interpreter. Some knowledge of LISP and PROLOG will be helpful in understanding the concepts in EqL.

This document is organized as follows: section II describes the data objects of EqL and the informal meaning of EqL rules. Section III describes interaction with the interpreter and consulting files. Section IV forms the heart of the document, and presents several examples of EqL programs for functional and logic programming, illustrating nondeterminism, logical variables, deferred evaluation of primitives, higher-order operations, and user-defined data constructors. Section V describes input/output, section VI debugging, and section VII other miscellaneous features of the interpreter.

EqL is an experimental language, and is likely to evolve as we gain more experience with it—desirable extensions in the language and its implementation are already planned. We therefore refer to the current language as 'EqL Version 1.0'. Our convention throughout this document is to use type-writer font, e.g. `cons`, for EqL program text, and italics, e.g. *equation*, for syntactic categories.

II. EqL Language Features

In this section we describe the data objects in EqL and the informal meaning of rules.

II.1. Data Objects

The data objects in EqL are defined below:

(i) *Numbers*: The current implementation of EqL provides only integers, e.g. `10, 207, 0, -11, -3999`, etc.

(ii) *Booleans*: `true, false`.

(iii) *Atoms*: Any identifier beginning with an upper-case letter or any sequence of characters enclosed within single quotes, e.g. `Apple`, `EqL`, `'also an atom'`, etc.

(iv) *Variables*: Normally begin with a lower-case letter, e.g. `x`, `y`, `tree`, `p1`, `q1`, etc. "Anonymous" variables begin with the underscore symbol, and serve as place holders in data structures, e.g. `cons(h, _dontcare)`. The underscore symbol by itself is also an anonymous variable.

(v) *Structures*: In the current implementation of EqL, there are two built-in structured data objects: *trees* and *strings*. User-defined structures may be specified using the `constructor` declaration, explained later. We explain trees and strings below.

As in LISP, the built-in constructor `cons(x,y)` defines a binary tree, and the operations `car(t)` and `cdr(t)` access the left- and right-subtree of a tree `t` respectively. Examples:

```
cons(10, 20),
cons(10, cons(20,30)),
cons(cons(a,10), cons(B, 20))
```

In the third example, note that `a` is a variable, but `B` is a atom.

Because list-processing is a common application of functional and logic languages, EqL provides a special notation for *lists*. Similar to lists in LISP and PROLOG, EqL lists are a special case of trees; they correspond to trees which "slope to the right" and end with the special symbol `[]`. The following examples illustrate the connection between lists and trees:

List Notation	*Tree Notation*
`[1,2,3]`	`cons(1,cons(2,cons(3,[])))`
`[[[1]]]`	`cons(cons(cons(1,[]),[]),[])`
`[]`	`[]`

Similar to PROLOG, the notation

```
[ x | y ]
```

is used to stand for `cons(x,y)`.

String constants are defined by a sequence of characters enclosed within a pair of double-quotes, e.g. `"abc"`, `"123"`, `"longer string"`, etc. The empty string is `""`. Analogous to the above notation for lists, we use

```
[ x :  y ]
```

to refer to the string obtained by prepending the one-character atom denoted by variable x in front of string denoted by y. For example, `['a' : "bc"]` is the string `"abc"`, and `['a' : ""]` is the string `"a"`.

We use the word *term* to refer to any data object of EqL that is built up from the above entities. We sometimes use the word *structured term* to refer to a term that has a constructor at the outermost level.

II.2. Rules

We illustrate program rules through examples. Below is a program to find the maximum depth of a binary tree of integers.

```
depth(x) => 0 where numberp(x) = true
depth([left | right]) => if dl > dr then dl+1 else dr+1
                         where
                                     dl = depth(left);
                                     dr = depth(right).
```

Program II.1: Maximum depth of a binary tree

The above program has two rules, which define the two cases for the depth function: the case for a leaf (an integer), and the case for a nonleaf (a tree built up from `cons`). In general, an EqL rule may take one of two forms:

1. *f(patterns)* `=>` *expression*.
2. *f(patterns)* `=>` *expression* `where` *equations*.

We explain each component of a rule below:

(i) *Patterns*. The word *pattern* is a synonym for *term*, which was defined in the previous section. Two or more patterns in a sequence are separated by commas, e.g.,

```
f([x1 | y1], [x2 :  y2], z) => ...
```

Zero-argument operations are permitted, and are defined by

```
f() => ...
```

(ii) *Expressions.* All expressions are evaluated in "applicative order," that is, leftmost-innermost expression first. The different kinds of primitives in the current implementation are listed below:

a. Terms: *atoms, numbers, booleans, variables* and *structures.*

b. Arithmetic: `+`, `-`, `*`, `/`, `div`, `mod`, and unary `-`. The operators `+` and `-` have lower precedence than `*` and `/`, which in turn have lower precedence than `div` and `mod`. Unary `-` has the highest precedence. The operators `/` and `div` both return the integer quotient. All binary operators are left associative. The function `abs(x)` returns the absolute value of x.

c. Relational: `<`, `>`, `<=`, and `>=`. The equality symbol `=` is reserved for defining equations. The function `eq(x,y)` maps identical atoms, numbers, and booleans to `true`; otherwise it returns `false`. The functions `lessp(x,y)`, `greaterp(x,y)`, `lesseq(x,y)`, and `greatereq(x,y)` are identical to `<`, `>`, `<=`, and `>=` respectively.

d. Boolean: `and`, `or` and `not`. The operator `not` has precedence over `and`, which has precedence over `or`. The following domain predicates are provided in the language.

`numberp(x)`	: `true` if x is a number; `false` otherwise.
`boolean(x)`	: `true` if x is a boolean; `false` otherwise.
`atom(x)`	: `true` if x is an atom; `false` otherwise.
`var(x)`	: `true` if x is an unbound variable; `false` otherwise.
`listp(x)`	: `true` if x is a list; `false` otherwise.
`null(x)`	: `eq(x,[])`.

e. If then else:

`if` *boolean-expr* `then` *expr1* `else` *expr2*

returns *expr1* if *boolean-expr* reduces to `true`, and *expr2* if *boolean-expr* reduces to `false`. Both the then- and the else-part may have equations associated with them. Thus, for example,

(*expr1* `where` *equations*)

may be used in place of *expr1* above. We explain *equations* further below.

f. Application:

`f`(*arguments*)

where `f` is the name of some built-in or user-defined function, or a variable that is bound to such a function, and *arguments* is a sequence of zero or more expressions separated by commas.

g. Input:

`read`(*filespec*)

returns the next EqL data item in the file specified by *filespec*. The `read` operation is discussed in detail in Chapter IV.

h. Output:

`write`(*filespec*, $e_1, \ldots, e_n$)

evaluates e_1 through e_n and writes their values in the file specified by *filespec* It returns the value of e_n, the last argument. The `write` operation is discussed in detail in Chapter IV.

(iii) *Equations*. An equation is of the form

$$expression_1 = expression_2.$$

However, for the purpose of explaining how equations are solved, we assume that an equation is of the form:

$$term = expression \qquad \text{or} \qquad expression = term.$$

Although permitted by our implementation, an equation $expression_1 = expression_2$ offers no extra power since it is equivalent to the pair of equations: v= $expression_1$; v = $expression_2$, where v is a distinct variable. Actually, it suffices to permit equations of the form v = *expression*, but permitting a *term* in place of variable v often leads to fewer intermediate variables in the source program, and hence clearer definitions.

Two or more equations in a sequence are separated by semi-colons. Because the interpreter is sequential, it solves equations in the sequence presented. There are two notable exceptions:

(1) An equation composed of terms, i.e., without any primitive or user-defined operations, would be solved prior to other types of equations. This reordering will not affect the correctness of the program's answer, and can help in avoiding unnecessary nontermination.

(2) An equation involving one of the primitive operators, such as `x = y + z` or `atom(x) = false`, would be deferred until sufficient information becomes

available to solve the equation. This delaying is discussed in greater detail in section IV.2.

In the equation-solving rules described informally below, we refer to expressions being 'evaluated', but the reader should note that an expression e is evaluated by solving an equation $v = e$. We assume henceforth that an equation is of the form *term* = *expression*; the symmetric case is treated similarly. There are six cases for an equation, corresponding to the six forms of an *expression*:

1. $term_1 = term_2$. This is solved by syntactic unification. The equation is unsatisfiable if unification fails, and initiates backtracking.

2. $term = f(e_1, \ldots, e_n)$, where each e_i is an *expression*, and f is a user-defined operation, whose first rule is

   $f(t_1, \ldots, t_n)$ `=>` *expression* `where` *equations*.

   The equation reduces to the following sequence of equations, where each v_i is a distinct variable:

   $$v_1 = e_1;\ \ldots\ ; v_n = e_n; \quad v_1 = t_1;\ \ldots;\ v_n = t_n; \quad equations; \quad term = expression.$$

   That is, the expressions e_1 ... e_n are first evaluated and their results then unified with t_1 ... t_n respectively; the *equations* in the body of f are then solved; and finally, the *expression* in the body of f is evaluated and unified with *term*. If at any stage unification fails, backtracking occurs to the dynamically most recent operation having an untried rule. Note that the evaluation order is essentially *call-by-value*. Also, the implementation ensures that each e_i is not re-evaluated when an alternate rule for f is attempted upon backtracking.

3. *term* = `if` p `then` e_1 `else` e_2. If expression p evaluates to a boolean, the equation $term = e_1$ or $term = e_2$ is solved. If p evaluates to an unbound variable, say x, two nondeterministic paths arise: (1) the equation $term = e_1$ is to be solved, where x $\leftarrow$ `true`; and (2) the equation $term = e_2$ is to be solved, where x $\leftarrow$ `false`. These two paths are tried out sequentially, with backtracking. Note that if e_1 or e_2 were accompanied by equations, these equations would be solved *before* solving $term = e_1$ or $term = e_2$.

4. *term* = *primitive*. The arguments of the primitive operator are first evaluated. If they are fully instantiated, the operator is applied to the arguments to produce a result,

which is then unified with *term*. Otherwise, the equation is *deferred*, or *delayed*, until all unbound variables are fully defined.

5. *term = read(filespec)*. The next EqL term is read from the file *filespec* and unified with the left hand side term. The equation is unsatisfiable if unification fails, and initiates backtracking.

6. *term = write(filespec, $e_1, \ldots, e_n$)*. This equation is equivalent to the following sequence of equations:

   $$v_1 = e_1; \ \ldots \ ; v_n = e_n; \quad term = write(filespec, v_1, \ldots, v_n).$$

   The value returned is that of v_n. This value is unified with *term*; the equation is unsatisfiable, and initiates backtracking, if unification fails.

Finally, in any place where an equation might appear, EqL permits a membership assertion of the form:

$$term \in expression$$

The above membership is satisfiable if *term* is unifiable with some object in the set of objects that expression evaluates to. The remaining objects, if any, are immediately discarded from further consideration. This construct is analogous to the "cut" in Prolog, but with cleaner semantics. We explain ∈ in greater detail in section IV.1.1.1.

III. The EqL Interpreter

The top-level query of an EqL program is either an expression or an equation or a set of equations, terminated by a period. A top-level expression e is actually treated internally as an equation, $\_ = e$, where _ is the anonymous variable.

A typical session in EqL is a "conversation" between the user and the interpreter. The interpreter is first invoked by the command

```
% eql
```

where we assume % is the Unix command-level prompt. The interpreter would respond as follows:

```
EqL Version 1.0

eql>
```

For example, the response to a query

```
eql> 2+4.
```

would be

```
6

eql>
```

To exit EqL, type CTRL-d when the above prompt appears. The interpreter will respond with

```
[ EqL execution halted ]

%
```

III.1 Consulting Files

Often, a set of EqL rules are kept in a file, which can be read in by a `consult` operation (as in C-Prolog). For example, if `depth` is the name of a Unix file containing the two rules for the `depth` function, it can be read in as follows.

```
eql> consult('depth').
```

EqL will respond with

```
true

eql>
```

Now, the `depth` function can be invoked on some input tree, e.g., `[10 | [20 | [30 | 40]]]`, as follows:

```
eql> depth([10 | [20 | [30 | 40]]]).
```

The interpreter would respond with

```
3

eql>
```

The `consult` operation can be used in any EqL rule, and will have the effect of reading in the rules contained in the file specified as the argument of `consult`. The `consult` returns true if the specified file is found, otherwise it returns false. Note that the rules from the specified file are read in only after the query evaluation is over and not at the time of the evaluation of the `consult` in the query. Note that `consult` augments the set of rules currently known to the interpreter; it does not replace any rule. Replacement of existing rules can be accomplished through the operation `reconsult`, described below.

Suppose that, after the file depth has been read in, the function depth is to be modified, say, to accept both atoms and numbers at the leaf of a tree. This could be achieved as follows: Use CTRL-z to suspend the interpreter; then edit the file depth so that the first rule reads as follows:

```
depth(x) => 0 where numberp(x) or atom(x) = true.
```

Re-enter the interpreter using the Unix foreground command, and re-consult the changed file by typing:

```
eql> reconsult('depth').
```

The new rules in the file depth will replace the old rules, and the interpreter will respond with:

```
true

eql>
```

In general, rules read in by re-consulting a file would replace all existing rules that define the same operations as those defined in the rules read in.

The user can input rules directly, without suspending the interpreter, by executing:

```
eql> consult('tty').
```

After receiving the prompt,

```
true

|
```

the user can type in one or more rules. The end of the input is specified with a period on a new line. This mode of supplying rules is sometimes convenient when the rules are short.

The interpreter may also be initially invoked by specifying any number of input files in the command line, e.g.

```
% eql -f file1 file2 file3
```

A query may be included in a file by preceding it with a ?. Queries may be placed at the beginning of a file, between rules, or at the end of a file.

Before we proceed further, we note again that the top-level query can, in general, be an equation or a sequence of equations. For example, the following goal is equivalent to depth([10 | [20 | [30 | 40]]]):

```
eql> x = depth([10 | [20 | [30 | 40]]]).
```

The interpreter's response to this goal would be:

```
    x = 3

    eql>
```

Actually, the top-level query `depth([10 | [20 | [30 | 40]]])` would in fact be internally converted into an equation, `_ = depth([10 | [20 | [30 | 40]]])`.

If the EqL interpreter finds that it cannot solve the top-level query, it would respond with the message,

```
    no solution
```

This might happen, for example, when the top-level query is:

```
    eql> depth([]).
```

because `[]` is not an atom. To account for this case, an explicit rule,

```
    depth([]) => 0.
```

can be provided.

III.2. Obtaining Multiple Solutions

Consider the following definition for the familiar append function of LISP, for non-destructively concatenating two lists:

```
    append([], x) => x.
    append([h|t], y) => [h | append(t,y)].
```

Program III.1: List append

For example, the result of the query

```
    eql> append([1, 2], [3, 4]).
```

would be the list

```
    [1, 2, 3, 4].
```

It is just as easy to find out the lists x and y such that when appended together will yield the list `[1,2,3,4]`. This query can be expressed as follows:

```
    eql> append(x,y) = [1,2,3,4].
```

The interpreter will respond with:

```
x = []
y = [1, 2, 3, 4]
```

Upon typing a semi-colon at the end of the second line above, the interpreter will respond with the second solution:

```
x = [1]
y = [2, 3, 4]
```

Typing a carriage return instead of a semi-colon will cause the interpreter to discard remaining solutions and return with the

```
eql>
```

prompt. All five solutions to the above query can be inspected by typing a semi-colon at the end of each preceding solution.

In general, when a semi-colon is typed and there are no further solutions, the interpreter will respond with the message:

```
no solution
eql>
```

The interpreter for EqL, like a PROLOG interpreter, explores alternative solutions to a query by depth-first search with backtracking.

IV. Programming in EqL

We now illustrate through examples the various features of EqL: non-determinism, delayed evaluation of primitives, logical variables, higher order operations, and user-defined constructors.

IV.1. Nondeterminism

We already saw in the previous chapter that multiple solutions may exist for an equation. We illustrate nondeterminism in EqL further through two examples: the family database, and the N Queens problem.

IV.1.1. Family Database

Shown below are four operations: `f(x)`, which returns the father of some person x; `m(x)`, which returns the mother of x; `p(x)`, which returns the parent of x; and `gp(x)`, which returns the grand-parent of x.

```
f(Bob)    => Gary.
m(Bob)    => Mary.
f(Ann)    => Gary.
m(Ann)    => Mary.
f(Gary)   => Joe.
m(Gary)   => Jane.
f(Mary)   => Steve.
m(Mary)   => Sue.
m(Joe)    => Donna.
m(Jane)   => Diane.
m(Steve)  => Diane.

p(x)      => f(x).
p(x)      => m(x).
gp(x)     => p(p(x)).
```

Program IV.1 : Family Relationships

The operation `p(x)` is nondeterministic because there are two rules for operation p. This reflects the fact that a person in this database has more than one parent—two to be precise. Because `gp(x)` is defined in terms of `p(x)`, it is easy to see that `gp(x)` is also nondeterministic. The grand-parent of some person, say `Bob`, could be found by:

```
eql> gp(Bob).
```

The first answer produced by the interpreter would be `Joe`, because `p(Bob)` would first return `Gary`, and `p(Gary)` would first return `Joe`. Upon requesting the next answer (by typing semi-colon), the interpreter would backtrack to the latest point where another choice is possible. Thus `p(Gary)` is recomputed, via `m(x)`, to be `Jane`, which becomes the next answer to the top-level query. The other answers, `Steve` and `Sue`, are determined similarly.

The grand-children of some person, say `Joe`, could be found with query such as:

```
eql> gp(x) = Joe.
```

IV.1.1.1. The use of <-

Suppose that we wanted to find the siblings of some person. Let us assume that two people are siblings if they have the same parents, and it suffices to check that they have one parent in common. We might initially try the following rule:

```
    sib(x) => y where p(x) = p(y); eq(x,y) = false.
```

With this definition of sibling, the response to a goal,

```
    eql> sib(Bob).
```

would first be Ann. Upon requesting another solution, we would once again receive Ann as the answer. The reason for this behavior is that the same answer, Ann, is discovered in two ways, first via the father of Bob, and again via the mother of Bob. We can avoid this behavior by selecting one parent of x, and then making sure that this parent is the same as that for y. In order to select an element from a set, the element-of construct, <-, can be used. The desired sibling definition is as follows:

```
      sib(x) => y where z <- p(x);
                              z = p(y); eq(x,y) = false.
```

Whenever the element-of construct is used to discard alternative solutions, the user should be aware that the "bi-directionality" of the operation using this construct could be affected. For example, a goal such as

```
    eql> sib(x) = y.
```

will not enumerate all possible sibling pairs; rather, it will produce at most one answer, depending upon whether the first parent found in the database had two children or not. In our example, the response to the above query will be

```
    x = Bob
    y = Ann
```

Upon typing a semi-colon, the interpreter would respond

```
    no solution
    eql>
```

indicating thereby that there are no further solutions.

IV.1.2. N Queens Problem

We now present a more complex example of nondeterministic programming. The problem is to place N queens on an N by N chess board in such a way that no two queens are attacking one another. A simple approach to this problem is to place queens on successive columns so that each new queen placed is not attacked by any queen in the preceding columns. If a queen cannot be placed on a given column, we go back to the preceding column to see if the queen there has another "safe" position. If there are no safe positions remaining on that column, we back up to the preceding column, and so on. A solution is found if we can thus place queens on all N columns. We have exhausted all solutions if we attempt to go back from the first column to the 0-th column.

Below is an EqL program for specifying the desired search:

```
queens(n) => solve (1, [], n).
solve(col, safelist, n) => if eq(col,n+1) then safelist
                              else place ([col | row(n)], safelist, n).
place(q, safelist, n) => solve (col+1, [q|safelist], n)
                                   where q = [col|row]:
                                       safe(safelist, q) = true.
safe([],q) => true.
safe([q1 | t], q) => safe(t,q) where threatened(q, q1) = false.
threatened([c1|r1], [c2|r2]) => eq(r1,r2) or eq(abs(r1-r2), abs(c1-c2)).
row(n) => n              where n>0 = true.
row(n) => row(n-1)       where n>0 = true.
```

Program IV.2 : N Queens Problem

The nondeterminism in the above program lies in the operation row(n), which generates the sequence of integers n, n-1, ..., 1, one at a time. This provides a way for stepping through the rows in any particular column.

IV.2. Delayed Evaluation of Primitives

An interesting aspect of the execution of an EqL program is the way primitive operators are handled. Basically, when a primitive operation, such as +, atom(x), >, etc., has all of its arguments defined, it is simplified to produce a result. However, when its arguments are not sufficiently defined when first encountered, it will be deferred until sufficient information is available to simplify it. This delayed evaluation has many uses, as we will illustrate in this and subsequent sections.

IV.2.1. Simulating Sets with Lists

The following rules define the familiar LISP operation member, which tests if an element x is a member of a list.

```
member([],    x) => false.
member([x|t],x) => true.
member([y|t],x) => member(t,x) where eq(x,y) = false.
```

Program IV.3 : List Membership

It is easy to see that member will correctly check if an element, say 3, is a member of some list, say, [1,2,3,4,5]. Member could just as easily be used to enumerate the elements of a list, using a goal such as

```
eql> member([1,2,3,4,5], z) = true.
```

which would return 1, 2, 3, 4, and 5 as the value for z, one at a time. What would happen if the goal

```
eql> member([1,2,3,4,5], z) = false.
```

were presented to the interpreter? The first rule of member fails because [] does not match [1,2,3,4,5]. The second rule initially succeeds in unifying the goal arguments with its patterns, but its result, true, fails to match false in the top-level query. Thus the third rule of member is taken.

When the operation eq(x,y) is encountered, y is bound to 1 but x is unbound. The EqL interpreter therefore defers this equation (by moving it to a global stack of such deferred equations), and proceeds with the recursive call on member. Each succeeding call on member results in one new deferred equation, eq(z,2) = false, eq(z, 3) = false, etc. Finally, the first argument to member is [], and rule 1 succeeds, and all equations are solved, except for the deferred equations. Because z is still unbound, the EqL interpreter responds as follows:

```
Input equations not fully constrained
z = _16
eql>
```

The binding of z to a number preceded by the under-score symbol indicates that z is unbound. Although the above behavior might not seem useful at first, consider the following natural definition of set difference (in terms of member).

```
diff(x,y) => d where member(y, d) = false;
                     member(x, d) = true.
```

Program IV.4 : Set Difference

The above rule states that the difference of two sets x and y (represented as lists) consists of elements d such that d is not a member of y and d is member of x. For example, the goal

```
eql> diff([1,2,3,4,5], [1,3,5,6,7]).
```

will return 2 and 4 as answers, one at a time.

EqL is able to find these answers as follows. At the end of solving the first equation, `member(y, d) = false`, where y = `[1,3,5,6,7]`, there will be five deferred equations:

```
eq(d,1) = false; eq(d,3) = false; ...  ; eq(d,7) = false.
```

When attempting solve `member(x, d) = true`, with x = `[1,2,3,4,5]`, the binding of d to the values `1, 3, or 5` will cause one of the deferred equations to fail, and hence these are determined not to be solutions. However, the binding of d to the values 2 or 4 will cause all the deferred equations to succeed, and hence these are determined to be solutions.

IV.2.2. Arithmetic primitives

Deferring the evaluation of arithmetic primitives also has some interesting uses. Consider the conversion of centigrade to fahrenheit, expressed by the following rule:

```
f(c) => 32 + 9*c/5.
```

This rule can be used to convert centigrade to fahrenheit by a goal such as

```
eql> f(100).
```

It can also be used to find the centigrade for some particular fahrenheit, by a goal such as

```
eql> f(x) = 212.
```

To understand the process by which this equation is solved, it should noted that EqL converts the above rule into the following program:

```
f(c) => 32 + t2 where t1 = 9*c; t2 = t1/5.
```

The top-level goal, `f(x) = 212`, results the following sequence of equations to be solved:

```
t1 = 9*x; t2 = t1/5; 32 + t2 = 212.
```

EqL defers the first two equations, and solves the last equation to obtain the value of t2. With this information t1 is determined from the second equation, and finally x from the first equation. Note that EqL will solve equations of the form $c = a\ op\ b$ where *op* is an arithmetic operator (+, -, *, /) and two of the three arguments (a, b, c) are known. With this capability, the reader may verify that the interpreter will be able to find the x such that

```
eql> fact(x) = 24.
```

where fact is the familiar factorial function, defined as:

```
fact(0) => 1.
fact(n) => n*fact(n-1) where n>0 = true.
```

We also leave it to the reader to explain why, for example, fact(x) = 23 will fail to terminate.

IV.3. Logical Variables

Much of the power in logic programming lies in its "logical variables." All variables in EqL are logical variables in that they derive their values not by direct binding but by the satisfaction of constraints. The following examples will serve as illustrations.

IV.3.1. Difference Lists

A classic example of logical variables is its use in defining "difference lists", which permit list concatenation in constant-time. The following is the EqL definition of difference-list concatenation.

```
dconc([x|t], [t|y]) => [x|y].
```

Difference lists can be used to avoid the use of append in many places. Consider, for example, the following inefficient definition of quick-sort.

```
qsort ([]) => [].
qsort([p|l]) => append(qsort(a), [p|qsort(b)])
                    where [a|b] = part(l, p).
part([], p) => [[] | []].
part([h|t], p) => if p>h then [[h|a] | b] else [a | [h|b]]
                    where
```

```
                                        [a|b] = part(t, p).
```
Program IV.5 : Naive Quicksort

The above program is inefficient because it employs a linear-time append operation at each stage of the sorting process. This inefficiency can be avoided by concatenating the sorted lists in constant-time by representing them as difference lists, as follows.

```
sort(l)            => answer where [answer | []] = dsort(l).
dsort([])          => [x|x].
dsort([p | l])     => [sorta | tail] where
                                    [a | b] = part(l,p);
                                    [sorta | [p| sortb]] = dsort(a);
                                    [sortb | tail] = dsort(b).
```
Program IV.6 : Quicksort with difference lists

IV.4. Higher-order operations

One of the advantages of functional languages over logic language is their support of higher-order functions. A well-known example is LISP's map function, which "maps" a unary function to each element of a list in order to produce a new list with mapped elements. This function can be expressed in EqL as follows:

```
map([], f) => [].
map([h|t], f) => [f(h)  |  map(t,f)].
```
Program IV.7 : List Mapping

For example, the goal

```
eql> map([0,2,3,4], 'fact').
```

would return the list [1,2,6,24] as the answer.

(As an aside, the reader may note that, because of deferred evaluation of arithmetic operators, the goal

```
eql> map([a,b,c,d], 'fact') = [1,2,6,24]
```

would yield a=0, b=2, c=3, and d=4!)

Because function names are atoms, they can be held in data structures. The following example shows how to take advantage of this ability.

```
simplify([op, e1, e2]) => op(simplify(e1), simplify(e2)).
simplify(n) => n where numberp(n) = true.

add(x,y) => x+y.
times(x,y) => x*y.
sub(x,y) => x-y.
quo(x,y) => x div y.
```

Program IV.8 : Simplification

For example, the goal

```
eql> simplify(['times', ['add', 3, 4], 10]).
```

would yield 70.

Note that EqL does not permit nested definitions of rules or global variables. All functions exist at one level; functions do not have any lexically-scoped or dynamically-scoped environment from which they obtain information. Just as functions can take function names as arguments, they may also be returned as results.

IV.5 Strings and User-defined Structures

Consider the following definition for non-destructively concatenating two strings:

```
cat([], x) => x.
cat([h:t], y) => [h :  cat(t,y)].
```

Program IV.9 : String Concatenation

Using the above definition, we may determine the two portions, `front` and `back`, of some string `s`, such that they are separated by some specific word `w`, as follows:

```
split(w, s) => [front, back] where cat(front, cat(w, back)) = s.
```

User-defined constructors are declared before their use. They can be declared anywhere in a file between the rules, e.g.

```
constructor:  seq, fun.
```

where `seq` and `fun` are the names of the constructors. These two constructors may be used in a program for performing type inference—`seq` and `fun` stand for sequence and function respectively. For example, the following rules illustrate how these constructors may be used to define the types of some primitive functions on sequences: `Nil`, `Head`, `Tail`, and `Adds`.

```
type(Nil) => seq(x).

type(Head) => fun(seq(x), x).

type(Tail) => fun(seq(x), seq(x)).

type(Adds) => fun(x, fun(seq(x), seq(x))).
```

IV.6. Programming Hints

To conclude this chapter we offer a few suggestions to aid the construction of more efficient program. These are not meant to be rigid rules, but general guidelines.

- Try to distinguish the different rules defining an operation based on the first argument of each rule—this facilitates *rule indexing* [G87], an optimization which eliminates unproductive alternatives by inspecting the first argument of rules.
- Use pattern matching instead of if-then-else — this leads to clearer programs and faster execution.
- Ensure that operands of strict operators are fully instantiated when they are to be applied — equation delaying is an expensive operation.

V. Input/Output

EqL provides two forms of I/O operations: non-backtrackable I/O, and backtrackable I/O. We first discuss non-backtrackable I/O.

To read data from standard input, use `'tty'` or `TTY` instead of the file name. The operation

`read(`*filespec*`)`

returns the next EqL data item in the file specified by *filespec*. This data item could be an atom, number, boolean, or any structured term, including trees, lists and strings. The operation

`read(line,` *filespec*`)`

would read the EqL items (which may be numbers or atoms) on the current line, and return a list of these items. The operation `read(list,` *filespec*`)`

would look for a list as the next item in the input. The operation

`read(char,` *filespec*`)`

would return the next character in the input, as an atom. The operations `read(int,` *filespec*`)` and `read(boolean,` *filespec*`)` allow reading an integer and boolean respectively.

All `read` operations would fail if an unexpected item is found in the input stream.

The `write` operation can take any number of arguments, which can be expressions. The expression

`write(`*filespec*, e_1, ..., e_n`)`

is treated as

`write(`t_1, ..., t_n`) where` $t_1 = e_1; \ldots; t_n = e_n;$

where each t_i is a distinct variable. The value returned by `write` is that of its last argument expression, namely, t_n. The non-graphic characters newline and tab are written following the C language convention, namely, `'\n'` (newline) and `'\t'` (tab).

EqL allows, as a special exception, a `write` to be used as an expression in any place where an equation might occur, e.g.,

```
write(TTY, 'Type a statement ending with period.', '\n')
```

This expression is converted internally into the equation:

```
_ = write(TTY, 'Type a statement ending with period.', '\n')
```

Because the current implementation is sequential, `read` and `write` operations have backtrackable variants, which, in this implementation, can occur only with respect to `'tty'`. To specify backtrackable read and write, use `readb` instead of `read`, and `writeb` instead of `write`, and omit the *filespec* in both cases. Backtracking past a `readb` would cause the data item read to be returned back to the input. Similarly, backtracking past a `writeb` would cause the output to be retracted. All output produced with a `writeb` will appear only when the top-level query has succeeded—the implementation maintains a buffer for all intermediate output; a similar buffer is maintained for backtrackable input.

The interpreter automatically opens a file when I/O is to be performed with it. At the end of each query, all files opened during the query are closed.

VI. Errors and Debugging

The EqL interpreter detects a variety of errors. The parser detects all syntactic and lexical errors. A few runtime errors, such as divide by zero, undefined operation, etc. are also detected. Debugging facilities have been built into the interpreter to help detect other sources of programming error.

The EqL parser reports the line numbers on which errors occurred and the token on the line near which the error occurred. No further analysis of the error is made. Usual causes of errors are: unmatched parentheses and brackets; omitting important punctuation, such as commas and semicolons; and using a reserved word as a variable name. The reserved words are as follows:

`true, false, abs, div, mod, and, or, not, if, then, else, where, read, write, readb, writeb, line, list, char, int, consult, reconsult, trace, constructor, cons, car, cdr, eq, atom, numberp, listp, var, boolean, null, lessp, greaterp, lesseq, greatereq, cputime, timer, save.`

The keyword `save`, for saving the execution state, has not yet been implemented.

One of the most common sources of run-time error, during initial program development, is the invocation of an undefined operation, or an operation with incorrect number of arguments. The interpreter will fail when this happens, and backtrack as usual, but will print out a message indicating that it did not find the desired operation.

Errors in logic can often be detected using the trace feature. Two forms of tracing are available: non-selective tracing, and selective tracing. The former is specified by

```
    eql> trace.
```

which prints out a trace of every call of every function. To turn off the trace, the same command is issued. To selectively trace a function `foo`, type

```
    eql> trace('foo').
```

and to turn off the tracing of `foo`, repeat the above command.

Because the interpreter performs *last equation optimization* [G87], which is a generalization of tail-recursion optimization, exit information for an operation may not always be printed out. We illustrate the trace feature for the following program, which computes (naively) the reverse of a list:

```
    rev([])    => [].
    rev([h|t])=> app(rev(t), [h]).
```

Suppose the top-level query were:

```
    eql> rev([1, 2, 3]).
```

The trace would be as follows:

```
trying rev at frame 0 with :    [1, 2, 3]
trying rev at frame 1 with :    [2, 3]
trying rev at frame 2 with :    [3]
trying rev at frame 3 with :    [ ]
Exiting frame 3
trying app at frame 2 with :    [ ]       [3]
Exiting frame 2
trying app at frame 1 with :    [3 ]      [2 ]
trying app at frame 1 with :    [ ]       [2 ]
Exiting frame 1
trying app at frame 0 with :    [3, 2]                 [1]
trying app at frame 0 with :    [2]       [1]
trying app at frame 0 with :    [ ]       [1]

[3, 2, 1]
eql>
```

Notice that all invocations of the append operation (see program III.1) are performed on the same frame, and hence no exit information for these operations is printed. Also, because of *rule-indexing*, unproductive choice points are discarded based on the arguments to rev and app. A trace feature showing a fuller account of the execution has not been installed in this version of the interpreter. The above information, nevertheless, is of much use in identifying errors in the program.

Errors in the interpreter would result in one of the following types of messages:

```
    ...   ·  panic
    Bus error:  ...
    Segmentation violation:  ...
```

The latter message is also produced when attempting to unify two infinite objects or printing out an infinite object.

VII. Other Interpreter Features

Timing:

The current time, in microseconds, is obtained from the variable cputime. To time the execution of a goal, such as rev([1,2,3,4,5,6,7,8,9,10]), one may write the following query:

```
eql> before = cputime;
     answer = rev([1,2,3,4,5,6,7,8,9,10]);
     time   = cputime - before.
```

The response, on a Sun-2, might be something like:

```
before = 2516666
answer = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
time = 150000
```

indicating that the run-time for the rev(...) goal was approximately .15 Sun-2 seconds.

The automatic timing of goals can be requested by the command

```
eql> timer.
```

which gives the cpu-time for all successive top-level queries, until turned off explicitly by the same command.

Interrupt:

The execution of any goal can be interrupted while in progress. The interpreter will trap the interrupt and respond as follows:

```
What now?  (type h for help):
```

Upon typing h, the response would be:

```
type a for abort
type c for continue
type t for trace
type u for untrace
type r for reset
type e for erase constructors
What now?  (type h for help):
```

Typing a causes the current query to be aborted; c continues execution; t starts tracing; u stop tracing; r causes all rules to be discarded; and e causes all constructor declarations to be wiped out.

Interpreter options

As mentioned in section III.2, the EqL interpreter may be invoked on any number of input files using the -f option. The other options allow the user to specify the sizes for various internal run-time data-structures except the *equation trail stack* which is implemented as a part of the *trail stack*. The general form of the Unix command line is

```
% eql [ -c control-stack-size ]
      [ -e equation-delay-stack-size ]
      [ -f file-names ]
      [ -h static-area-size ]
      [ -r read-stack-size ]
      [ -t trail-stack ]
      [ -v variable-stack-size ]
      [ -w write-stack-size ]
```

The *control-stack* holds the control information for each invocation of a user-defined operation; the space for variables is allocated separately in a *variable-stack*. Except for numbers and booleans, each entry in the variable-stack is a pointer. For structured objects, this pointer points to a *molecule-heap*. As in Prolog implementations, the *trail-stack* is used to record variables whose binding must be undone upon backtracking. The *read-stack* and *write-stack* are used to implement backtrackable read and write operations respectively. The *equation-delay-stack* is used to implement delayed execution of equations. The default allocations, measured in terms of stack entries, for each of the above data structures are: 75000 (`-c`), 10000 (`-e`), 75000 (`-h`), 10000 (`-r`), 75000 (`-t`), 150000 (`-v`), and 10000 (`-w`).

Implementation Note:

EqL has been implemented by Gopal Gupta as part of his Master's Thesis [G87]. The implementation has been carried out under the Unix operating system, using the tools Lex, Yacc, and C. Planned enhancements include: set expressions, garbage collection, and modules. EqL can be obtained, at nominal distribution cost, by writing to the first author, whose e-mail address is `bj@cs.unc.edu`, or `...!decvax!mcnc!unc!bj`.

References

[J85] B. Jayaraman, *Equational Programming: A Unifying Approach to Functional and*

Logic Programming, TR 85-030, Dept of Computer Science, University of North Carolina at Chapel Hill, October 1985.

[JS86] B. Jayaraman and F.S.K. Silbermann, Equations, Sets, and Reduction Semantics for Functional and Logic Programming, In *1986 ACM Conference on LISP and Functional Programming*, M.I.T., August 1986, pp. 320-331.

[J87] B. Jayaraman, *Semantics of EqL*, To appear in *IEEE Transactions on Software Engineering*, 1987.

[G87] G. Gupta, *An interpreter for EqL*, M.S. Thesis, Dept. of Computer Science, University of North Carolina at Chapel Hill, expected December 1987.